

A Lock-Free Multiprocessor OS Kernel

Henry Massalin and Calton Pu
Columbia University
June 1991

Presented by: Kenny Graunke
<kennyg@cs.pdx.edu>

Where are we?

- Just to ground ourselves.....welcome to 1991
- Previously - Synthesis V.0
 - Uniprocessor (Motorola 68020)
 - No virtual memory
- 1991 - Synthesis V.1
 - Dual 68030s, virtual memory, supports threads
 - A significant step forward
 - A fairly ground-up OS

Recap: The problem

- Roughly...two threads could process the same data at the same time...
 - Inconsistent or corrupted data
 - Crashes (dangling pointers, double frees, ...)
 - Obnoxiously hard to debug (race conditions)
- Most OSes solve this with mutual exclusion
 - a.k.a. locking

Problems with Locks

- Locking can be trouble
 - Granularity decisions
 - Increasingly poor performance (superscalar; delays)
 - Hard to compose...
 - Which locks might I be holding?
 - Is it safe to call some code while holding those locks?
 - Deadlock...
 - Crash while holding a lock...?

Alternative Approach

- No locks at all.
- Use lock-free, “optimistic” synchronization
- Goal: Show that Lock-Free synchronization is...
 - Sufficient for all OS synchronization needs
 - Practical
 - High performance
- But what is it?

“Pessimistic” Synchronization

- Murphy's law: “If it can go wrong, it will...”
- In our case:
 - “If we can have a race condition, we will...”
 - “If another thread could mess us up, it will...”
- Hide the resources behind locked doors
- Make everyone wait 'til we're done
 - That is...if there was anyone at all

Optimistic Synchronization

- The common case is often little/no contention
- Do we really need to shut out the whole world?
- If there's little contention, there's no starvation
 - Lock-free instead of wait-free
- Small critical sections really help performance

Optimistic Synchronization

1. Write down any preconditions/save state
2. Do the computation
3. Atomically commit results:
 - Compare saved assumptions about the world with the actual state of the world
 - If different, discard work, start over with new state
 - If preconditions still hold...store results, complete!

Stack Code

```
Pop() {  
    retry:  
        old_SP = SP;  
        new_SP = old_SP + 1;  
        elem = *old_SP;  
        if (CAS(old_SP, new_SP, &SP) == FAIL)  
            goto retry;  
    return elem;  
}
```

Stack Code (Clarification)

```
Pop() {  
    retry:  
    old_SP = SP;  
    new_SP = old_SP + 1;  
    elem = *old_SP;  
    if (CAS(old_SP, new_SP, &SP) == FAIL)  
        goto retry;  
    return elem;  
}
```

local variables – can't change underneath us

“global” (at least to the data structure)
...other threads can mutate this at any time

Stack Code (Stages)

```
Pop() {  
    retry:  
    I old_SP = SP;  
    I new_SP = old_SP + 1;  
    I elem = *old_SP;  
    I if (CAS(old_SP, new_SP, &SP) == FAIL)  
        goto retry;  
    return elem;  
}
```

- I 1. Write down preconditions
- I 2. Do computation
- I 3. Commit results (or retry)

Optimistic Synchronization

- Specifically in this paper...
 - Saved state is only one or two words
 - Commit is done via Compare-and-Swap (CAS) or Double-Compare-and-Swap (CAS2 or DCAS)
- Wait, only two words?!
 - Yep! They think they can solve all of the OS's synchronization problems while only needing to atomically touch two words at a time.
 - Impressive...

Approach

- Build data structures that “work” concurrently
 - Stacks, Queues (array-based to avoid allocations)
 - Linked lists
- Then build the OS around these data structures
 - Concurrency is a first-class concern

If all else fails... (the cop out)

- Create a single “server” thread for the task
- Callers then...
 1. Pack the requested operation into a message
 2. Send it to the server (using lock-free queues)
 3. Wait for a response/callback/...
- The queue effectively serializes the operations

Double-Compare-and-Swap (DCAS)

an atomic instruction

```
CAS2(old1, old2, new1, new2, mem_addr1, mem_addr2) {  
    if (*mem_addr1 == old1 && *mem_addr2 == old2) {  
        *mem_addr1 = new1;  
        *mem_addr2 = new2;  
        return SUCCEEDED;  
    }  
    return FAIL;  
}
```

Stack Push (with DCAS)

```
Push(elem) {
```

```
  retry:
```

```
    I old_SP = SP;
```

```
    I new_SP = old_SP - 1;
```

```
    I old_val = *new_SP;
```

```
    I if (CAS2(old_SP, old_val, new_SP, elem, &SP, new_SP) == FAIL)
```

```
        goto retry;
```

```
}
```

I 1. Write down preconditions

I 2. Do computation

I 3. Commit results (or retry)

old_val is useless garbage!
To do two stores, we must do two compares.

Comparison with Spinlocks

```
Pop() {  
    spin_lock(&lock);  
  
    elem = *SP;  
    SP = SP + 1;  
    spin_unlock(&lock);  
    return elem;  
}
```

```
Pop() {  
    retry:  
    old_SP = SP;  
    new_SP = old_SP + 1;  
    elem = *old_SP;  
    if (CAS(old_SP, new_SP, &SP) == FAIL)  
        goto retry;  
    return elem;  
}
```

Performance Analysis

(Taken from Massalin's dissertation)

Operation	Non Sync	Locked	Lock-free _{noretry}	Lock-free _{oneretry}
null procedure call in C	1.4	—	—	—
Increment counter	0.3	2.4	1.3	2.3
Linked-list Insert	0.6	2.7	1.4	2.3
Linked-list Delete	1.0	3.2	2.1	4.1
Circular-Queue Insert	2.0	4.2	3.3	6.0
Circular-Queue Delete	2.1	4.3	3.3	6.0
Stack Push	0.7	2.8	2.0	3.5
Stack Pop	0.7	2.8	2.0	3.5

Times in microseconds, measured on a 25Mhz 68030, cold cache

Impact

- This paper spawned a lot of research on DCAS
 - Improved lock-free algorithms
 - The utility of DCAS
- DCAS is not supported on modern hardware
 - “DCAS is not a Silver Bullet for Nonblocking Algorithm Design”

Conclusions

- Optimistic synchronization is effective...
 - ...when contention is low
 - ...when critical sections are small
 - ...as locking costs go up
- It's possible to build an entire OS without locks
- Optimistic techniques are still applicable
 - ...though the implementation (DCAS) is not